Efficient Distribution of Security Policy Filtering Rules in Software Defined Networks

Ahmad Abboud, Rémi Garcia, Abdelkader Lahmadi, Michaël Rusinowitch, Adel Bouhoula

Efficient Distribution of Security Policy Filtering Rules in Software Defined Networks

Ahmad Abboud[†*], Rémi Garcia[‡*], Abdelkader Lahmadi[*], Michaël Rusinowitch[*], Adel Bouhoula[§†]
[*] Université de Lorraine, CNRS, Inria, Loria, F-54000 Nancy, France, {firstname.lastname}@inria.fr
[†] NUMERYX, France, a.abboud@numeryx.fr, a.bouhoula@numeryx.fr
[‡] IPB Enseirb-Matmeca, France, rgarcia003@enseirb-matmeca.fr
[§] College of Graduate Studies, Arabian Gulf University, P.O. Box 26671, Kingdom of Bahrain, a.bouhoula@agu.edu.bh

***Abstract*—Software Defined Networks administrators can specify and smoothly deploy abstract network-wide policies, and then the controller acting as a central authority implements them in the flow tables of the network switches. The rule sets of these policies are specified in the forwarding tables, which are usually accessed using very expensive and power-hungry ternary content-addressable memory (*TCAM*). Consequently, a given table can only contain a limited number of rules. However, various applications need large rule sets to perform filtering on diverse flows. In this paper, we propose several algorithms for decomposing and distributing a rule set on network switches of limited flow tables size, while preserving the network policy semantics. Through experiments on several rule sets with single and multiple dimensions, we evaluate and analyse the performance of our rule placement techniques. Our results show that our proposals are efficient in practice.**

***Index Terms*—Packet filtering, Software defined networks, Rule placement**

I. Introduction

In software-defined networks (SDN), the filtering requirements for critical applications often vary according to flow changes and security policies. SDN address this issue with a flexible software abstraction, allowing simultaneous and convenient modification and implementation of a network policy on flow-based switches. This single-point deployment approach constitutes a key feature for complex network management operations.

The growing number of attacks coming from diverse sources continually increasing the number of entries in access-control lists (ACL). To avoid relying on large and expensive memory capacities in network switches, a complementary approach to rule compression [1], [2], [3] would be to use multiple smaller switch tables to enforce in the network the access-control policies. It paves the way towards distributed access-control policies, which have been the topic of many previous studies [4], [5], [6]. However, most of them have a large rules replication [4], [5] or even they modify the header of the packet [6] to avoid the matching of a rule by a packet in the next switch.

In this work, we tackle the problem of rule-placement of ACLs, while focusing on three major challenges:

- Simplicity and efficiency of rule set decomposition to facilitate security policy deployment and updating;
- Decomposition over diverse network topologies;
- Decomposition of complex network policies.

In this paper we introduce new techniques to decompose and distribute filtering rule sets over a given network topology. Our approach is to design decomposition schemes for both simple and complex policies relying on a single or multiple dimensions, leading to multi-level network architectures to forward packets to the destination.

Like OBS [4] and Palette [5], we rely on rule dependency relations, but we generate less forward rules by taking the action field into consideration. Indeed, a packet is allowed to be matched by rules with two prefixes from successive switches, if their resulting action does not violate the initial policy semantics.

In the following sections, we introduce efficient decomposition and distribution algorithms for network policy management as well as the decomposition possibilities related to different topologies. Section II introduces some related work on the rule placement problem. In Section III, we state the general principles and distribution constraints in this problem. In Section IV, we perform a forward table decomposition using the commonly used Longest Prefix Matching (*LPM*) priority strategy in OpenFlow [7] wildcard matching. Section V details the distribution of rule tables with respect to relative positions of neighbour devices. In Section VI, we generalize the decomposition problem with filtering on multiple dimensions and arbitrary rule priority strategies. Section VII presents benchmarks and comparisons with the previous frameworks [4], [5], [6], using their and our own rule sets. We present our conclusions in Section VIII.

II. Related work

The decomposition of rules can be specific to application requirements regarding the matching fields. Some work aims to reduce the total number of rules used after the decomposition whereas others try to minimize energy consumption or maximize traffic satisfaction [8].

The type of rule sets used in each approach has a significant impact on the decomposition. Normally, in order to match a packet with a set of rules we use two strategies either according to the order of rules in a prioritized list, or according to the longest matching prefix (LPM).

Using a prioritized list, OBS [4] tries to decompose rules using a two-dimensional endpoint policy. In their approach,

overlapping rules lead to their replication in multiple switches. An optimization for choosing the best coverage on two-dimension representation is proposed in [9], in order to improve the storage and the performance of the decomposition of rules. More optimizations have been done in [10], [11], in order to choose the best cut while minimizing the tables size, the rule duplication and the pre-processing time for rule updates.

Using another approach, Palette [5] proposes two methods for splitting a set of rules, the first one being called Pivot Bit Decomposition where a table can be divided in two sub-tables, by changing a wildcard bit "*" to 0 or 1 in each table. The second method called cut-based decomposition relies on a dependency graph to represent a set of rules. In this method each graph should be placed in the same switch, which can cause problems with large graphs due to a high number of overlapping rules.

A dependency graph has been introduced in [12], in order to divide rules into two parts for an OpenFlow environment. The first part is cached and the second is deployed while preserving the semantics of the policies.

A near-zero decomposition overhead technique has been proposed in [6]. This technique is based on adding an additional bit to the packet header. When a packet has matched a rule, this bit ensures that the packet will not be processed again. While costing only a bit per rule and in the packet, this stateful technique requires non standard modifications on the packet structure and leads to security issues when an attacker could manipulate this additional bit to bypass the filtering mechanism.

The rule distribution problem has been studied in [13] where a predefined set of rules can be shared by multiple paths across the network.

Our work shares with these previous works the problem of distributing a rule set among network switches to meet a policy. However, in our approach we rely on the properties of the LPM strategy and leverage action field information in order to handle overlapping rules and avoid replication in switch tables. Unlike [6] we do not apply packet or rule modifications. In addition, we also exploit the structure of a series-parallel graph to resolve efficiently the rule placement problem for all-sized networks in tractable time, unlike many existing approaches [13], [4], [14] that relies on Integer Linear Programming (ILP) techniques.

III. Problem statement

A. Problem definition

We consider a network N with SDN-enabled switches that are connected to each other using their respective ports. Each switch can store in its flow table different types of access-control and filtering rules. Each flow table has a limited capacity regarding the number of stored rules. We assume that an SDN policy is a collection of rules generated by an administrator or by an external module. The size of the set of rules of the policy exceeds the capacity of a single switch table.

In this paper, we are mainly concerned by decomposing and distributing a set of filtering rules along the switches in the network N to accomplish an SDN policy while preserving its overall semantics and meeting the capacity limitation of each switch. In the network N, packets are controlled by rules stored in their Forward Information Bases or flow tables. A rule is specified by a priority, matching patterns on packet fields and an action. A matching pattern of a source (resp. destination) field is given by a list of $0, 1$ followed by $*$'s (*don't care bits*), of global length w (word length) called a prefix p. Any bit matches wildcard character $*$. A rule with prefix p for source (resp. destination) applies to a packet if this packet source (resp. destination) field matches p, bit by bit. We call bit-prefix of the rule the sub-list of $0, 1$ of p.

The matching process for a given set of rules can follow different strategies. A simple strategy prioritizes the rules by their order in the rule set. A packet matching the first rule will apply the action of that rule without considering the following ones. With a Longest Prefix Matching (LPM) strategy, one packet can match multiple rules, but only the one with the most specific matching prefix (i.e., the longest prefix) will be selected. In the example shown in Table I, if a switch receives a packet with 0001 as address, and using a prioritized list strategy, action A_1 of the first rule is applied. However, with an LPM strategy, the packet matches both rules 1 and 2, but only action A_2 will be applied since rule 2 covers the address field with a longer prefix.

Rule	Address field				*Action*
1	0	0	*	*	A_1
2	0	0	0	*	A_2

TABLE I: Example of a rule set in a switch table.

B. Distribution and decomposition requirements

To ensure that distributing the rules along different switches does not change the initial policy compared to a single-switch placement, we must preserve its overall semantics, i.e., *the action applied on a matched packet in a single switch with the initial policy rule set should be the same action in a chain of switches with the distributed policy rule set.* Here, we consider that filtering rules have two possible actions: "Forward" or "Deny". In the following, the prefix of a rule r is denoted by $Pref(r)$ and its action by $Action(r)$. If $Pref(r_1)$ matches $Pref(r_2)$ and has a shorter bit-prefix than $Pref(r_2)$, like Rule 1 with Rule 2 in Table I, then we say that $r_1 > r_2$ or r_1 overlaps r_2. In the example of Table I, it is also true that $Pref(r_1)$ is the next longest matching prefix i.e there is no r such that $r_1 > r > r_2$. We express this by $r_1 >: r_2$.

Let R be the initially given rule set of a policy. Let R_1 and R_2 be two subsets of R located in different switches along a path, with R_2 being located in a switch after the one of R_1 as illustrated in Fig. 1.

When applying the LPM strategy in a switch, a packet must be processed by the most specific matching rule. Thus,

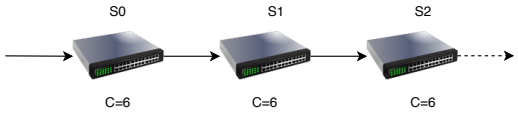


Fig. 1: Flow table capacities of switches along a single path.

if $r_2 \in R_2$, there must not exist any $r_1 \in R_1$ such as $r_1 > r_2$. To enforce this precedence property, when we place a rule r in a switch, we must also place in the same switch any rule r' such as $r > r'$.

When a packet has been accepted by a switch containing rule set R_1 and enters a switch containing rule set R_2, if r_1 and r_2 are matching rules in R_1 and R_2 respectively, and $r_2 > r_1$ then r_2 should not block it. Thus, we must prevent blocking a packet when there is an action conflict. Given two rules $r_1 \in R_1$ and $r_2 \in R_2$, we have an action conflict iff $r_2 >: r_1$, $Action(r_1) = "Forward"$ and $Action(r_2) = "Deny"$.

In this work, we rely on forward rules to solve action conflicts: if two conflicting rules r_1 and r_2 belong respectively to two successive switch tables R_1 and R_2, a rule fw with $Action(fw) = "Forward"$ and $Pref(fw) = Pref(r_1)$ is added to R_2. When the policy preserving condition is respected, $Pref(fw)$ has a higher specificity, thus priority, than $Pref(r_2)$ by construction.

When decomposing R into subsets to be stored in the different switches and adding forward rules to solve action conflicts, the decomposition problem is formulated as a Bin Packing problem with fragmentable items [15] where each fragmentation induces a cost.

IV. Decomposition over a path

Networks often have a blacklisting policy which specifies that packets originating from certain IPs are potentially harmful and need to be dropped, for instance when handling denial of service attacks [16]. Thus, in a first step, we resolve the decomposition problem with a single filtering field and by using an LPM strategy, which could be sufficient for placing a blacklisting policy in the network.

A. Rule representation

We use a binary tree to represent the rules ordered by their prefix specificity. We study the single field case, so each rule contains one filtering field and there is at most one rule associated to a tree node. The sequence of edge labels taken from the root to a node represents the bit-prefix of the rule linked to this node.

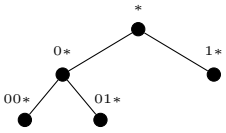


Fig. 2: Compact representation of rules prefixes in a binary tree.

As shown in Fig. 2, the pattern $00*$ of a rule r is at distance 2 from the root, reachable through the leftmost branch of the tree.

The complete prefix with wildcards of a node n represents a rule pattern. It is denoted as $NodePrefix(n)$ and is equal to $Pref(r)$, when a rule r is contained in n. We denote by $fw(n)$ the forward rule with a matching pattern $NodePrefix(n)$.

Once the rules are distributed among the switches, if in any switch no matching rule is found for an incoming packet, the packet will be forwarded to the next switch using a default rule.

B. Forward rules generation

We assume the set of rules R has been distributed along successive switches $s_1, \ldots s_n$ forming a path in the network graph. For this distribution to be correct with respect to the initial R and LPM semantics, we need to introduce additional forward rules in some switches.

The forward rule generation is illustrated in Fig. 3, where we install half of a rules in the first switch.

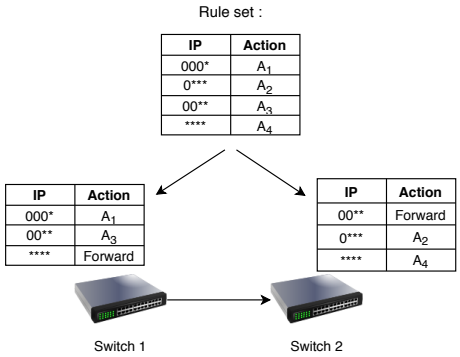


Fig. 3: Illustration of forward rule generation with a rule set and two successive switches.

For the first switch, we have selected rules with 00** as common prefix, so accepted packets with an IP matching this pattern must be accepted in the second switch. Thus, we have generated a forward rule with prefix 00** in the second switch. A default forward rule has also to be added to the first switch to allow the second one to receive any unmatched packets.

Given a set of rules R, two rule prefixes $p_1 = w0*, p_2 = w1*$ can be merged to obtain a forward rule prefix $p' = w*$. This operation can be iterated.

Fig. 4 illustrates a rule tree with merging possibilities. This tree contains 4 rules with prefixes $00*, 01*, 1*, 10*$. We iteratively merge $00*$ and $01*$ to form a prefix $0*$, which is merged then with $1*$ into a common forward prefix. In this example, only one forward rule is needed for this set. As shown in Fig. 4, the forward rule covers the subtree with rules at every branch above the blue line.

The set of resulting prefixes obtained by iterating the merging operation in R and that are maximal for $>$ is called $MaxFwdMerges(R)$.

On a filtering path of length n, let R_i be a rule subset placed in switch $s_i, 0 < i < n$. The potentially necessary

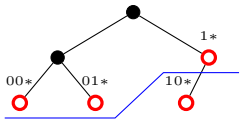


Fig. 4: Single-forward rule tree.

forward rules in $s_{i+1}, i+1 \leq n$ after adding rules in s_i is composed of successive $MaxFwdMerges$ results and called $PotentialFwds$. Considering R_{i+1}, only forward rules needed to resolve action conflicts with $R_{1,\ldots,i}$ are added in s_{i+1}. Thus, the forward rule set needed in switch s_{i+1} is a subset of $PotentialFwds$. If a prefix covers only deny rules in R, packets matching this prefix in s_i will not travel to the next switch. Such a prefix is not added to $PotentialFwds$ of s_{i+1} since there will be no packet to match.

C. Decomposition algorithm

Let us consider the binary tree shown in Fig. 5 which represents an ordered rule set. As an illustration example, each node contains one rule and we need to decompose the rule set along the path shown in Fig. 1. In this path, each switch has a maximal capacity of 6 rules. If we take into account the default rule in each switch, we need to find a set of 5 rules in order to completely fill the first switch. As depicted in Fig. 5, we select the set of nodes containing a total of 5 rules maximum (all nodes in red). After this selection step, all rules added to the switch will be removed from the binary tree. If some space remains in the switch, we will try to find other rules candidate.

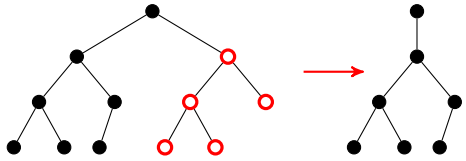


Fig. 5: Choosing a set of candidate rules from the binary tree to be placed in a switch.

The rule distribution is performed in one pass following a bin-focused approach and a **Minimum Bin Slack** heuristic [17]. Thus, each switch is filled as much as possible with a set of rules while trying to minimize the overhead caused by action conflicts with rules in previous switches. The required extra forward rules in switch i for a rule set R_i is defined as $NeededFwds(R_i, PotentialFwds) = \{fwd \in PotentialFwds \mid \exists r \in R_i$ such that $r >: fwd \wedge Action(r) =$ "$Deny$"$\}$.

Algo. 1 is firstly called with an empty $PotentialFwds$ and it describes the decomposition of a given rule set over switches in a single path.

The function $ChosenCandidates$ searches in the rules tree R for a rule subset R_i that can fit in switch s_i, and has a maximal $|R_i| - |NeededFwds(R_i, PotentialFwds)|$ value.

Algorithm 1 DecPath($R, \langle s_1, s_2, \ldots, s_k \rangle, PotentialFwds$)
Input: set of rules R, k switches s_i of capacities c_i, set of forward rules $PotentialFwds$
Output: $ChosenRules$, the rules added to the k switches;
R, the remaining rules;
$PotentialFwds$, the forward rules computed after adding $ChosenRules$.

```
1: Let ChosenRules ← ∅
2: for i = 1 to k do
3:     Add a default forward rule to s_i
4:     Let R_i ← ChosenCandidates(R, s_i, PotentialFwds)
5:     ChosenRules ← ChosenRules ∪ R_i
6:     Add R_i ∪ NeededFwds(R_i, PotentialFwds) to s_i, replacing the
       default forward rule if R_i contains the default rule
7:     PotentialFwds ← MaxFwdMerges(R_i ∪ PotentialFwds)
8:     R ← R \ R_i
9: end for
10: return ⟨R, PotentialFwds, ChosenRules⟩
```

D. Algorithmic complexity

When building the rules tree, we can efficiently add information that will speed up the set selection part. That way, we record with every node i) the number of rules in its subtree, ii) whether this subtree is a blacklist, and iii) whether this node prefix is a valid result of prefix merges. When a rule r is inserted in a given node t_n, every parent of t_n can update these variables in $O(w)$ since w (word length) is an upper bound for the length of the bit-prefix of r. The rule tree construction part is then achieved in $O(nw)$ time.

Selecting the rules to be stored in a switch requires at most an entire tree traversal in $O(nw)$ time. Given a node prefix p, searching for a rule conflict with it can be done in $O(w)$ time, as the forward rules can also be represented in a tree data-structure. The decomposition can be done in $O(nw)$ time for every switch.

We adopt a greedy approach where the potential forward rules tree representation of $O(nw)$ size is the main runtime memory cost.

V. Decomposition over a graph

In this section, we present a technique to enforce the same filtering policy across all paths in the network. We present our algorithm for distributing a set of rules on a network whose topology is a two-terminal series-parallel graph [18]. This algorithm is not specific to LPM, it can be used on all types of rule matching strategies and with any dimensions. Then we will show how to handle more general two-terminal directed acyclic graphs.

A. Decomposition over a two-terminal series-parallel graph

Series-parallel graph are widely used in telecommunication networks [19], since the failure of a network part can be mitigated by using a parallel path, especially in critical applications with low tolerance to network paths failure. These types of networks can be updated in order to easily add or remove some parts by dividing the network into parallel or series compositions.

Definition 1. *A two-terminal directed acyclic graph (st-dag) has only one source s and only one sink t. A series-parallel(SP) graph is an st-dag defined recursively as follows:*
(i) A single edge (u, v) forms a series-parallel graph with source u and sink v.
(ii) If G1 and G2 are series-parallel graphs, so is the graph obtained by either of the following :
(a) Parallel composition: identify the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2. (b) Series composition: identify the sink of G_1 with the source of G_2.

A two-terminal series-parallel graph is a graph with distinct source and destination vertices, and obtained by a set of series and parallel compositions, merging the source and destination of components in case of series decomposition or by merging the two sources and the two destinations in case of parallel decomposition as shown respectively in Fig. 6 and Fig. 7.

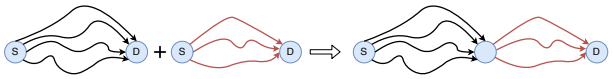


Fig. 6: Series composition.

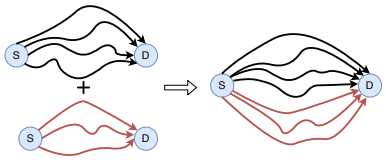


Fig. 7: Parallel composition.

A series-parallel graph can be represented by a binary tree [20], where each internal node of the tree is a series or a parallel composition operation and each leaf is an edge of the graph. Fig. 8 shows a binary tree representation of a series-parallel graph. We define an S-component as an oriented path, that is either an edge or a series composition of edges (see Fig. 9). We can derive a more compact representation of series-parallel graphs by replacing maximal subtrees built solely from series operators by the S-components they represent. Hence, leaves are S-components in this alternative representation.

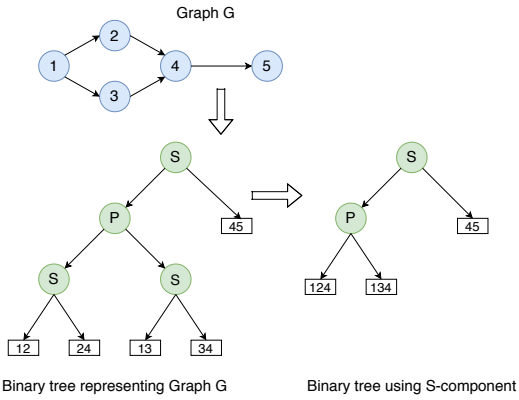


Fig. 8: Tree representation of a series-parallel graph.

In the example of Fig. 8, edges $1 \rightarrow 2$ and $2 \rightarrow 4$ can be merged into one S-component, as edges $1 \rightarrow 3$ and $3 \rightarrow 4$.
In order to determine if our algorithm can find a solution given

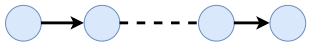


Fig. 9: S-component.

a rule set R to decompose, we need to try the decomposition on all paths of the directed network graph. We will then find the smallest rule set that can fit in all paths, taking into account the overhead in terms of forward rules and the compatibility between different paths that share some switches.

Algo. 2 outputs a rule set put in a specific path from s to t using a representation of the graph as a binary tree with S-components. Let R_0 be the initially given policy rule set, and N_0 the root of a binary tree T_0 representing the series-parallel network graph. Each node N of the binary tree will be labelled with a triplet $\langle R^{'}, F^{'}, R\rangle$, where :

$R^{'}$ is the remaining rules to place in next switches.
$F^{'}$ represents the $PotentialFwds$ set computed from $R_0 \setminus R^{'}$.
R is the subset of rules from R_0 occurring in the subtree rooted at N.

The obtained labelling can be interpreted as a solution of the distribution problem. We define the solution affected to a node n in T_0 to be the label of the subtree rooted at n.

Algorithm 2 DecGraph(N, R, F)
Input: N, node in T_0; R, set of rules;
F, set of forward rules to be installed in the following switches.
Output: label of N

```
1: if N.isS-Component then
2:     ⟨R', F', R⟩ ← DecPath(R, N.switches, F)        ▷refer to Algo. 1
3:     return ⟨R', F', R⟩
4: else if N.isSeries then
5:     ⟨R', F', leftR⟩ ← DecGraph(N.leftChild, R, F)
6:     ⟨R'', F'', rightR⟩ ← DecGraph(N.rightChild, R', F')
7:     return ⟨R'', F'', leftR ∪ rightR⟩
8: else if N.isParallel then
9:     ⟨R', F', leftR⟩ ← DecGraph(N.leftChild, R, F)
10:    ⟨R'', F'', rightR⟩ ← DecGraph(N.rightChild, R, F)
11:    while leftR ≠ rightR do
12:        if |leftR| < |rightR| then
13:            ⟨R'', F'', rightR⟩ ← DecGraph(N.rightChild, leftR, F)
14:        else
15:            ⟨R', F', leftR⟩ ← DecGraph(N.leftChild, rightR, F)
16:        end if
17:    end while
18:    return ⟨R', F', leftR⟩                          ▷equal to ⟨R'', F'', rightR⟩
19: end if
```

If node N corresponds to an S-component, the field $N.switches$ above is by definition the sequence of switches in the S-component minus the first one if that S-component is in series composition with a previous one.

Algo. 2 has as parameters a node of T_0, a set of rules $R \subseteq R_0$, a set of forward rules F. In the main program the recursive procedure Algo. 2 is called with N equal to the root of T_0, F

empty, R equal to R_0. Algo. 2 terminates successfully when at the end the first field R' of the root label is empty: this ensures that all rules have been distributed successfully and no rule has been left out.

This algorithm can be used with other decomposition algorithms like those defined in [4], [5], [6] just by modifying the *DecPath* function in line 2 with another one. Note that, in our approach and unlike the Optimized Big Switch (OBS) [4] technique, we do not create several rule table partitions for each path traversing an intersection switch. A packet will be processed by the same rule table at an intersection, regardless of the path it came from. This also facilitates policy updating, as the places where some specific rule occur are easier to localize.

B. Algorithmic complexity

The labels computed by Algo. 2 when successful, defines a solution to the rule distribution problem. The labels are obtained after exploring the entire graph, and the tree representation allows for a single processing of edges shared by several paths.

Several trial and error steps can be necessary for rule propagation, but in many cases the process is substantially faster, since only one propagation try has to be performed under a parallel node. Such cases include paths with switches of identical capacities, or rule sets in which action conflicts cannot happen, for example when the rules sets are blacklists. In this way, given a graph $G = (V, E)$, Algo. 2 computes a rule set decomposition and distribution in $O(p|E|)$ time where p is the number of parallel nodes. Since the cost for distributing rules on a switch is $O(nw)$, we have an overall complexity of $O(p|E|nw)$.

C. Generalization to st-dags

In Subsection V-A, we discussed how to distribute rule sets in series-parallel graph. Here, we show that this technique can be applied to arbitrary two-terminal directed acyclic graphs or st-dags. Duffin [18] proved that a two-terminal st-dag is series-parallel if and only if it does not contain any subgraph homeomorphic to the Braess graph in Fig. 10.

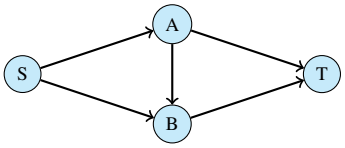


Fig. 10: The Braess graph.

We could notice in such a graph that having the same rule sets in vertices A and B still leads to a correct distribution: a packet travelling from A to B would be applied the same rule and action a second time. To generalize the distribution mechanism, A and B could then be considered as only one vertex, eliminating a Braess graph to obtain a series-parallel one. We then merge them as illustrated in Fig. 11.

Let $G = (V, E)$ the initial network graph, which is not series-parallel. Let a Braess component of G be a subgraph

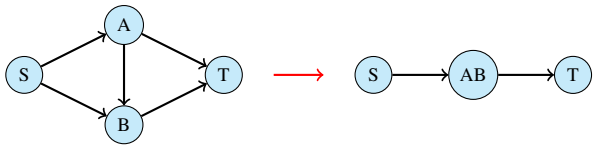


Fig. 11: Merging of vertices to eliminate Braess components.

whose vertices are labeled as in Fig. 10. Let us define the set $Inter$ to be the set of vertices occurring on any path from A to B including A and B. We perform a merge on A and B to obtain the graph $G' = (V', E')$ defined as follows:

- V': $(V \setminus Inter) \cup \{AB\}$ with $capacity(AB) = min(capacity(A), capacity(B))$
- E':
  - if $x \to y \in E \wedge x = A \wedge y \notin Inter$ then $AB \to y \in E'$
  - if $x \to y \in E \wedge x = B$ then $AB \to y \in E'$
  - if $x \to y \in E \wedge y \in \{A, B\}$ then $x \to AB \in E'$
  - if $x \to y \in E \wedge \{x, y\} \cap Inter = \emptyset$ then $x \to y \in E'$

Iterating the operation and once every Braess component has been eliminated, we apply our distribution algorithms on the resulting series-parallel graph. The decomposition and distribution is valid as shown by:

Lemma 1. *If we have a solution for the distribution problem of the initial rule set R_0 in G', then we can construct a solution for the graph G.*

Proof: Leaving empty the intermediate vertices between A and B, we know exactly where the rules will be located. As AB contains a rule set which can fit in both A and B by definition, this rule set is duplicated in A and B. The solution in G' is supposed valid, so precedence and intersection constraints at AB are satisfied. Thus, paths $S \to \cdots \to A$ and $S \to \cdots \to B$ contain the same rules from R_0 and the potentially necessary forward rule set Fw is the same just after A and B. For the same reason, it is also true that paths $A \to \cdots \to T$ and $B \to \cdots \to T$ contain the same rules from R_0 and generate the same forward rules. Distribution constraints are then satisfied for paths $S \to \cdots \to T$ and the solution affected to G is valid. □

By induction, we conclude that if we can build a solution for a graph obtained by eliminating successively Braess graphs then we can construct a solution for the initial st-dag.

VI. A TWO-TIER APPROACH FOR MULTI-FIELDS RULE SETS

In our first approach described in sections IV and V, we restricted the rule set as single field filtering rules, mainly to reduce the forward rules overhead for conflicting actions. Although this technique is useful for blacklisting policies, it is limited and becomes inefficient for decomposing and distributing complex policies that involve filtering rules with IPs and ports fields and overlapping.

If we considered for example both source and destination IP fields, we could get a large overhead in terms of forwarding

rules as shown in Fig. 12 since for each rule added in Switch 1 we need to generated a forward rule for it in the Switch 2. Unless we find mergeable prefixes on the second filtering field, we would generate as many rules as we have removed from the initial rule set, which limits the decomposition interest.

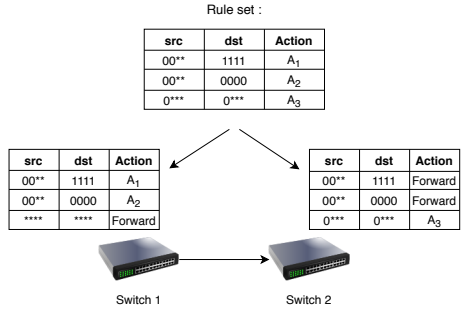


Fig. 12: A counter-productive example of forward rules generation when using our first approach with multi-field filtering rules.

It is nevertheless feasible to apply this approach to several fields when the rule set overlapping rules reside in the same switches or when we are dealing with a blacklist. With all actions of rules being "Deny", all packets matching some rules are stopped and the other are captured by the default rule to be tested in the next switch, so non-default forward rules are unnecessary in a blacklist case.

In [6], the authors propose a technique to deactivate rule matching on next switches when a packet is matched in the current switch. However, their technique requires a non-standard modification on the packet structure by adding an additional bit.

We believe that a more practical solution could be used in real-world environments, and avoiding modifications on the packet structure. To achieve this, we propose a novel approach with nearly zero rule space overhead in switch tables.

When distributing a rule set in a network, a packet coming from the source switch needs to travel along a path towards the destination. Only a single filtering action needs to be applied to a given packet. Thus, if a packet is accepted in a switch, all the next switches have to transfer the packet to the destination since a match with a higher priority rule has already been applied. In our first approach, the forward rules mechanism allows us to do that, but increases the total number of rules in the switches. To avoid this overhead problem and allow the distribution of multi-fields rule set, we use two types of switches: Forward Switches (FoS) and Filtering Switches (FiS) as shown in Fig. 13 .

All FiS at distance 2 or more from the destination are connected to one FoS, as it would be impractical to connect all switches to the destination in practice. The role of a FoS is to transmit a packet directly to the destination in order to skip the next FiS on the path, reducing the path length and network latency. For the sake of simplicity we use three types of actions in a rule: "Forward", "Deny" and "Default forward". If the selected action is "Forward", the packet is sent to a FoS, and if the action is "Deny", the packet is dropped. The default forward action sends the packet to the next filtering switch.

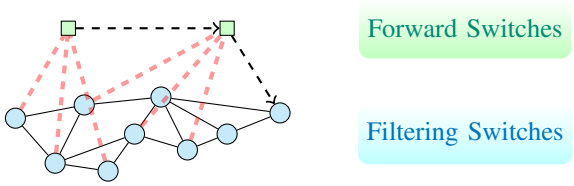


Fig. 13: Network topology using FoS and FiS.

By using two types of switches, in this second approach, we eliminate the need for non-default forward rules, which reduces the total number of rules used over a path, and accelerates the processing time of each packet by forwarding it to the destination directly using FoS. A practical requirement for this technique is to rely on high throughput FoS. This technique is applicable in any priority strategy, assuming that every switch which enforces the network policy is linked to the destination, directly or through FoS.

A suitable networking architecture for our approach could be the Flat-tree structure [21], used in datacenters. In a flat-tree architecture, we can perform matching tests over successive switches by using neighbor-to-neighbor connections and benefit from a multi-tier topology. Indeed, once a packet matches a rule, it can be forwarded directly to the top-level switch instead of going through the other filtering switches.

To distribute the rules on a network graph, the approach described in Section V can be applied without forward rules. Therefore the decomposition algorithm can be simplified, since the F' component of node label of the binary tree in Algo. 2 can be eliminated.

VII. Evaluation

A. Simulation setup

In our evaluation experiments of the different proposed algorithms, we rely on 12 rule sets available at [22]. These rule sets and ours are generated using the ClassBench tool. All our algorithms were implemented in Java with single-threaded programs. The machine used for this evaluation is a desktop computer with Intel core i7-7700 3.6-GHz CPU, 32 GB of RAM and running the last version of Windows 10 operating system.

We define the overhead OH in terms of additional forward rules placed on the switches of a path for a given rule set as follows:

$$OH = \frac{N_t - N_i}{N_i}$$

where

N_t is the total number of rules used in a path (initial rule set plus extra forward rules).

N_i is the number of rules in the initial rule set.

The overhead depends on the length of the path, capacity of the switches on the path and the diversity of rules' actions in the rule set.

B. Results of LPM based decomposition with forward rules

The 12 rule sets used in our evaluations are the same as in [6]. In this evaluation all switches have the same capacity (tables size), and the minimal capacity to find a solution by our distribution algorithm is defined as C_{min}. We decompose the rule sets using the source IP address as a filtering field. Even with a synthetic rule set of 64000 random rules, no rule set takes more than 150ms to be decomposed and distributed. This processing time includes the tree building, rule set decomposition and rules distribution. This level of performance should enable a network administrator to react and deploy almost immediately an updated policy suited to a new kind of attack flow.

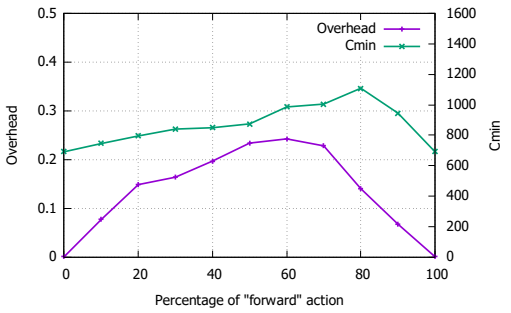
Applying our approach on two classical rule sets, which are whitelists and blacklists, the forward rules overhead is very low since such rule sets contain few actions conflicts. For example, with a whitelist policy, only the last default rule has a "Deny" action, so only the last switch will need extra rules in its table. In terms of a blacklist policy, no forward rules are required, so the tree structure is only useful as a sorting and updating mechanism. The decomposition of multi-fields filtering policies is possible when dealing with blacklists, with only one default forward rule added to each switch except the last one.

In our simulations, the action field is being used in order to study the effect of this field on the number of generated rules. If a set of rules R1 (0*,deny) and R2 (*,forward) need to be divided using OBS, a forward rule for R1 will be generated to avoid conflict with R2. But using our approach no forward rule should be generated since a packet matching R1 in a switch dose not continue to the next one.

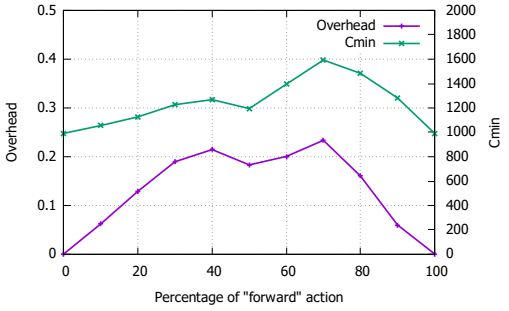
In order to show the effect of the action field on the overhead OH and the minimal capacity C_{min}, we use two rule sets (fw1 and acl1), from the 12 rule sets [6], with 0 to 100% percentage of rules having "Forward" action. The average values of OH and C_{min} are computed by running 10 simulations for each percentage value since the "Forward" actions will be distributed randomly among prefixes.

Fig. 14 shows the effect of the action field on both OH and C_{min} metrics. When a set of rules has very low action diversity, the overhead is very low since the number of conflicting actions is very small. The situation where the half of the rules has a "Forward" action introduces the most high overhead, and we obtain similar results with the other rule sets.

Fig. 15 shows the overhead distribution using the 12 rule sets from Classbench while varying the path length. The overhead in the worst case is around 30% with 8 switches and for 50% of accepting rules. By using LPM, our approach does not necessarily introduce rule duplication with sets having overlapping rules like the OBS approach. If a rule r is selected, no rule r' such as $r' > r$ is needed in the current switch. Only forward rules fw such as $r > fw$ are added to to resolve action conflicts. However, the performance of our LPM based method can not be compared to OBS in terms of placement



(a) fw1



(b) acl1

Fig. 14: Effect of action field on overhead OH and C_{min} using two different rule sets.

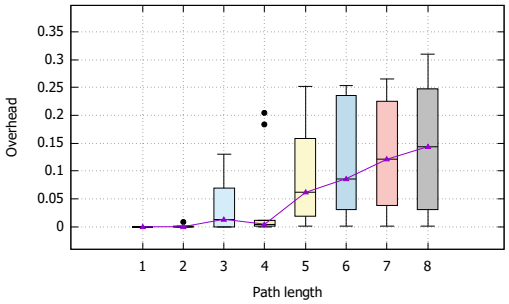


Fig. 15: Rule space overhead while distributiong a rule set with a 50% of rules having Forward actions.

results, because of the single filtering field that we use. The OBS approach operates on both source and destination fields, which is not the case for our method.

Contrary to Palette, we do not cut the packet header space by half, so our performance is not affected by the path length being or not a power of two as highlighted in OBS benchmarks. We also do not need to build classifiers such as rules from two different classifiers do not intersect. We only have to respect the rule precedence relation, which enables us to decompose a given rule set without the overhead of rule bits expansion. While Palette assumes that the classifiers have the same size at the end of the decomposition, our approach is not dependent on switch capacities, as each next switch receives as much rules as it can with our algorithm.

Our technique based on LPM is limited to rules composed of a bit-prefix followed by wildcards. For general rules with

wildcards possibly occurring at any position, we have to perform a rule expansion like in Palette Pivot Bit Decomposition [5].

C. Results of forward switches based approach

Our second two-tier approach for multi-fields rule sets does not introduce rule space overhead except for the extra default rule in each switch, and provides results similar to the OneBit [6] approach. However, our approach has low rule space overhead, it respects packets structure without introducing extra bits, and ensures in a stateless way that packets can perform a network traversal without being matched. Table II shows the different values of C_{min} obtained from [6], in addition to our two-tier approach results on the same datasets with a single path of 10 switches topology. Our approach similar to OneBit divides all rules over the path of 10 switches with an equivalent number of rules in each switch. In this case each switch will be $\sim$99% full. The only extra cost will be one default rule by switch. On the contrary, using Palette or OBS for example, and for a set of 9968 rule such as fw2, the value of C_{min} is around 4000. In this case the number of used switches belonging to the path will be low, but these approaches introduce the cost of using switches with higher capacity and higher overhead because of rules duplication which affect the performance of the matching rule process along the network.

TABLE II: Values of C_{min} for different approaches on 12 rule sets with a single path of 10 switches.

		Palette	OBS	ONEBIT	TWO-TIER
datasets	Size	C_{min}	C_{min}	C_{min}	C_{min}
acl1	9928	2480	1030	997	994
acl2	7433	2308	1214	746	745
acl3	9149	3622	2687	919	916
acl4	8059	2870	1706	809	807
acl5	9072	2265	912	910	909
fw1	8902	3776	2423	891	891
fw2	9968	4308	3688	997	998
fw3	8029	3877	2818	804	804
fw4	2633	1196	800	268	265
fw5	8136	2651	1780	814	815
ipc1	8338	2260	1088	837	835
ipc2	10000	4348	2406	1002	1001

Our two-tier approach can be used with arbitrary rules priority criterion and can handle rule sets with multiple filtering fields. In this approach no packet will be modified and the only added rules are the default ones in each switch. In addition, the forwarding switches (FoS) will decrease the time needed for a packet to reach it destination since a FoS does not perform any rule matching test. Additionally, as a FoS is a very simple forwarding device, the cost of adding a few FoS to the network remains reasonable.

VIII. Conclusion and future work

In this paper we propose a new decomposition technique based on LPM for one field. We then introduce a novel rule set distribution algorithm for series-parallel network graphs. Finally, we present a solution for multiple fields leveraging two levels of network switches in the SDN data plane. This solution also applies to other than LPM rule selection schemes. We conducted a series of experiments on rule sets generated from Classbench. Our results show reasonable rule space overhead in worst-case scenarios for LPM without Forward Switches, and close to zero overhead with our two-tier solution. Our future work consists of tackling the rule update strategy problem. We also plan to implement our solutions in real environments.

Acknowledgement

This work is supported by a CIFRE convention between the ANRT (National Association of Research and Technology) and the company NUMERYX Technologies.

References

[1] A. Abboud, A. Lahmadi, M. Rusinowitch, M. Couceiro, A. Bouhoulal, and M. Avadi, "Double mask: An efficient rule encoding for software defined networking," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 186–193.

[2] Y. Sun and M. S. Kim, "Tree-based minimization of TCAM entries for packet classification," in *2010 7th IEEE Consumer Communications and Networking Conference*, Jan 2010, pp. 1–5.

[3] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 18–30, Jan 2012.

[4] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," 12 2013, pp. 13–24.

[5] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 545–549.

[6] P. Chuprikov, K. Kogan, and S. Nikolenko, "How to implement complex policies on existing network infrastructure," in *Proceedings of the Symposium on SDN Research*, ser. SOSR 18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185477

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *Computer Communication Review*, vol. 38, pp. 69–74, 04 2008.

[8] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules placement problem in openflow networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 1–1, 12 2015.

[9] W. Li and X. Li, "Hybridcuts: A scheme combining decomposition and cutting for packet classification," in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, 2013, pp. 41–48.

[10] W. Li, X. Li, H. Li, and G. Xie, "Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2645–2653.

[11] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "Parasplit: A scalable architecture on fpga for terabit packet classification," in *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, 2012, pp. 1–8.

[12] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, 08 2014.

[13] P. Kannan, M. Chan, R. Ma, and E.-C. Chang, "Raptor: Scalable rule placement over multiple path in software defined networks," 06 2017, pp. 1–9.

[14] S. Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik, "An adaptable rule placement for software-defined networks," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 88–99.

[15] B. LeCun, T. Mautor, F. Quessette, and M.-A. Weisser, "Bin packing with fragmentable items: Presentation and approximations," *Theoretical Computer Science*, 01 2013.

[16] P. Ferguson and D. Senie, "Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing, bcp 38, rfc 2827," May 2000. [Online]. Available: https://www.rfc-editor.org/info/bcp38
[17] J. N. D. Gupta and J. C. Ho, "A new heuristic algorithm for the one-dimensional bin-packing problem," *Production Planning & Control*, vol. 10, no. 6, pp. 598–603, 1999.
[18] R. Duffin, "Topology of series-parallel networks," *Journal of Mathematical Analysis and Applications*. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022247X65901253
[19] P. Flocchini and F. L. Luccio, "Routing in series parallel networks," *Theory of Computing Systems*, 2003. [Online]. Available: https://doi.org/10.1007/s00224-002-1033-y
[20] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, ser. STOC 79. New York, NY, USA: Association for Computing Machinery, 1979, p. 112. [Online]. Available: https://doi.org/10.1145/800135.804393
[21] Y. Xia, X. Sun, S. Dzinamarira, D. Wu, X. Huang, and T. Ng, "A tale of two topologies: Exploring convertible data center network architectures with flat-tree," 08 2017, pp. 295–308.
[22] *Code for Palette, OBS and OneBit simulations*, 2017. [Online]. Available: https://github.com/distributedpolicies/submission

APPENDIX A
CORRECTNESS OF RULE DISTRIBUTION ON SERIES-PARALLEL GRAPHS

After applying Algo. 2 on the syntax tree of a series-parallel graph, the relations between the labels in each node are partially determined by which rule sets are put in subtrees with a parallel node root. Taking the example of Fig. 8, we can draw labels dependency relations.

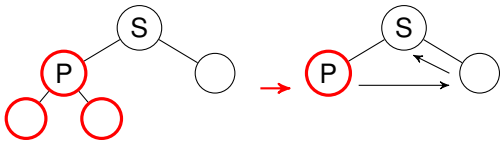


Fig. 16: Dependency path between subtrees' labels.

In Fig. 16, the children of the parallel node contain the same subset of rules from R_0, so they share a common label with their parent, which is denoted by their red color. The simplified tree at the right is the minimal one necessary to establish the final labels.

As stated, a subgraph G_2 labelling depends on the label affected to the subgraph G_1 whose switches lead to G_1's source. These labellings must summarize a correct LPM rule distribution.

Assuming that $DecPath$ implements the LPM semantics of the initial rule set R_0 in the path directly given, we show it is also true for the series composition of path components on which $DecPath$ is called:

Lemma 2. *Switches in a given path enforce a correct LPM policy.*

Proof: The first S-component encountered in a prefix traversal is constituted of the first switches in the path. Applying $DecPath$ on this component will ensure that LPM is respected among its switches. The recursive calls of Algo. 2 follow the prefix order, so the precedence relation between switches is maintained.

Lines 4-7 of Algo. 2 maintain the coherence between calls and resulting labellings under a series node. If N with label $\langle R', F', R\rangle$ has children, then for the left and right ones T_1 and T_2, if $T_1.L = \langle R'_1, F'_1, R_1\rangle$ then $T_2.L = \langle R'_2, F'_2, R_2\rangle$ with:

- $R' = R'_2 = R'_1 \setminus R_2$
- $F' = F'_2 = MaxFwdMerge(R_0 \setminus R'_2)$
- $R_2 \in R'_1$
- $R = R_1 \cup R_2$

With this precedence and forward relations, a series node's right child will receive the appropriate forward rules and rules from R_0 as base for distribution in its subtree.

$DecPath$ is called several times for different components of a path. The definition of $N.switches$ guarantees that in a series composition, the common vertex between two components is not filled twice. With a path $s_1 \rightarrow s_2 \rightarrow s_3$, $DecPath$ could then be called on $s1 \rightarrow s_2$ then s_3, so s_2 would not be filled again.

Consequently, the result of successive $DecPath$ calls on path components is equal to the one obtained with a single call on the whole path. This path contains a rule set decomposed with respect to the intended LPM policy. □

When a packet has the option to take different paths towards a network location, each of these paths must be equivalent, filtering-wise. Algo. 2 must terminate correctly in this case.

Lemma 3. *Switches in parallel paths enforce the same filtering policy.*

Proof: Lines 8-18 ensure that a parallel node will share the same labelling as its two children, i.e its children subtrees will contain the same rules from R_0. Algo. 2 always terminates, because of the process of decomposition trial and error. After decomposing a part of R_0 in the two parallel subgraphs represented by T_1 and T_2, we obtain R_1 and R_2 respectively fitting in those subgraphs. Three cases can occur:

- R_1 is identical to R_2.
- R_1 or R_2 has smaller size than the other.
- R_1 and R_2 are different rule sets of the same size.

These last two cases indicate a decomposition mismatch. To correct it, Algo. 2 iteratively tries to propagate the smallest rule set resulting from T_1 or T_2. If R_1 has to be propagated on T_2, $R'_1 \subseteq R1$ will fit in T_2. Because of this inclusion relation, R'_1 can fit in T_1 and Algo. 2 returns it once propagated. The process is symmetrical if R_2 has to be propagated on T_1. Thus, Algo. 2 terminates after having ensured that two parallel subgraphs enforce the same filtering policy. Only one set of forward rules can be used at the common sink of these subgraphs. □